Suporte a Modelos de Discussão na Engenharia orientada a Modelos com a Ontologia Kuaba

Support for Discussion Models in Model-Driven Engineering with the Kuaba Ontology

Luiz Eduardo dos Santos Cunha
Universidade Federal Fluminense
Rio das Ostras - RJ
eduardo_cunha@id.uff.br

João Victor de Souza Pimentel Cunha
Universidade Federal Fluminense
Rio das Ostras - RJ
jv_cunha@id.uff.br

Rafael Abrahão da Costa
Universidade Federal Fluminense
Rio das Ostras - RJ
rafael_abrahão@id.uff.br

Adriana Pereira de Medeiros
Universidade Federal Fluminense
Rio das Ostras - RJ
adrianamedeiros@id.uff.br

RESUMO

Pesquisas sobre desafios técnicos relacionados à incerteza durante a modelagem de software são fundamentais para a melhoria das soluções oferecidas na engenharia dirigida a modelos (MDE - *Model-Driven Engineering*). Essa incerteza é causada por muitas alternativas de *design*, informações incompletas e opiniões conflitantes dos *stakeholders* que geralmente não são consideradas nessas soluções. A abordagem Kuaba para *design rationale* trata as causas dessa incerteza, pois permite o registro e o processamento dessas informações durante a elaboração dos modelos. Este artigo propõe uma abordagem MDE que combina *design* dirigido a domínio (DDD) e a arquitetura orientada a modelos (MDA - *Model-Driven Architecture*) para conectar modelos de discussão gerados com Kuaba aos artefatos produzidos durante o *design* de software. A abordagem proposta permite lidar com a incerteza uma vez que considera o registro da discussão entre os *stakeholders* sobre o domínio que será apoiado pelo software. Além disso, viabiliza a geração dos modelos de domínio e de *design* a partir dessa discussão e posterior geração de código, utilizando transformações de modelos em uma ferramenta de modelagem de software.

Palavras-Chave

Design Rationale; Ontologia Kuaba; Engenharia orientada a Modelo; Arquitetura orientada a Modelos; *Design* Dirigido a Domínio; Ferramentas de *Design* de Software.

ABSTRACT

Research into the technical challenges related to uncertainty during software modeling is fundamental to improving the solutions offered in Model-Driven Engineering (MDE). This uncertainty is caused by many design alternatives, incomplete information, and conflicting stakeholders opinions that are often not considered in these solutions. The Kuaba approach to design rationale addresses the causes of this uncertainty, as it allows the recording and processing of this information during model development. This paper proposes an MDE approach that combines domain-driven design (DDD) and model-driven architecture (MDA) to connect discussion models generated with Kuaba and artifacts produced during software design. The proposed approach allows dealing with uncertainty since it considers the record of the discussion between stakeholders about the domain that will be supported by the software. Furthermore, it enables the generation of domain and design models from this discussion and subsequent code generation using model transformations in a software modeling tool.

Keywords

Design Rationale; Kuaba ontology; Model-Driven Engineering; Model-Driven Architecture; Domain Driven Design; Software Design Tools.

CCS Concepts

• Software and its engineering→Software creation and management→Software development techniques.

1. INTRODUÇÃO

Embora as pesquisas em engenharia orientada a modelos (MDE - *Model Driven Engineering*) [18] tenham alcançado progresso nos últimos anos, ainda há desafios técnicos a serem investigados para a melhoria das soluções oferecidas nessa área. Um desses desafios técnicos apresentado em [2] está relacionado à incerteza durante a modelagem de software, causada por muitas alternativas de *design*, informações incompletas e opiniões conflitantes dos *stakeholders*. Assim, de acordo com [2], uma das questões a serem consideradas é como usar a MDE para conectar modelos de discussão com artefatos de software. Outro aspecto comentado é a necessidade de alavancar/integrar ferramentas de modelagem flexíveis.

Todo software está relacionado a um domínio, que contém o conhecimento relacionado às atividades de interesse de seus usuários. Segundo Eric Evans [4], o coração do software é a habilidade de resolver os problemas do domínio para o usuário. Quando o domínio é complexo, os desenvolvedores precisam

estudá-lo para construir o conhecimento do negócio. No entanto, isso não é prioridade na maioria dos projetos de software. Geralmente os desenvolvedores não têm muito interesse em aprender sobre o domínio específico em que estão trabalhando. O domínio é confuso e exige novos conhecimentos que parecem não desenvolver as capacidades de um cientista da computação. É comum iniciarem o *design* sem conhecer bem o domínio para o qual estão desenvolvendo o software, ou seja, tentam resolver problemas de domínio com tecnologia [4]. Assim, a incerteza durante a modelagem geralmente tem o seu início na compreensão do domínio a ser apoiado pelo software.

Abordagens para a representação de *design rationale*, como a abordagem Kuaba [9] utilizada neste trabalho, podem ajudar a reduzir essa incerteza, uma vez que permitem estruturar o raciocínio utilizado durante a modelagem, registrando os problemas analisados, as alternativas de solução avaliadas para cada problema e os argumentos para a escolha de uma alternativa em detrimento de outra. Assim, o *design rationale* representa um modelo de discussão no qual toda a deliberação para a tomada de decisão durante a modelagem fica registrada.

Este artigo propõe uma abordagem MDE que combina o *design* orientado a domínio (DDD - *Domain Driven Design*) [4] e a arquitetura dirigida a modelo (MDA - *Model Driven Architecture*) [13] para conectar modelos de discussão, gerados com Kuaba, com artefatos produzidos durante o *design* de software. O artigo apresenta também o suporte desenvolvido em uma ferramenta de modelagem para apoiar o uso da abordagem proposta.

O uso da abordagem Kuaba e do DDD em soluções MDE auxilia a compreensão dos modelos utilizados na geração automática de código. Também favorece a colaboração entre os desenvolvedores, uma vez que permite a análise das possíveis soluções de *design* avaliadas em cada modelo, de acordo com o domínio, e a consulta aos prós e contras de cada solução, contribuindo, assim, para reduzir a incerteza durante a modelagem de software.

O restante deste artigo está organizado da seguinte forma: na seção 2 são apresentados conceitos fundamentais para a compreensão do trabalho; na seção 3 é apresentada a abordagem Kuaba-MDE; a seção 4 apresenta o suporte desenvolvido para o uso da abordagem em uma ferramenta de modelagem de software; na seção 5 é apresentado um exemplo de uso da abordagem na ferramenta; na seção 6 são discutidos alguns trabalhos relacionados; e a seção 7 apresenta as considerações finais sobre o trabalho.

2. REFERENCIAL TEÓRICO

O desenvolvimento de software é uma atividade realizada em equipes compostas por diferentes *stakeholders*. É comum existir dificuldades na definição das abstrações necessárias para atender as demandas do domínio. Assim, é importante fornecer suporte para capturar essas abstrações durante a discussão entre os desenvolvedores e especialistas de domínio. Essas abstrações fazem parte do *design rationale* do projeto, que descreve as razões por trás das decisões de *design*. Neste trabalho, um modelo de discussão é uma representação do *design rationale* capturado durante a compreensão do domínio a ser apoiado pelo software.

Kuaba é uma abordagem baseada em argumentação para representação de *design rationale* em *design*s baseados em modelos [9]. Ela utiliza uma ontologia de mesmo nome, a ontologia Kuaba [10], nessa representação.

A abordagem Kuaba usa a semântica fornecida pelo metamodelo utilizado no *design* para instanciar os elementos básicos dessa ontologia. A Figura 1 mostra uma visão geral da abordagem Kuaba.

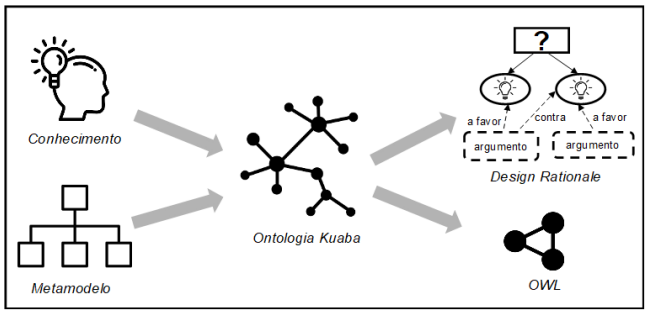

Figura 1. Abordagem Kuaba [8]

Em Kuaba, os itens de informação do domínio que está sendo modelado e os argumentos (conhecimento tácito) a favor ou contra as ideias consideradas durante a modelagem são fornecidos pelos especialistas de domínio ou desenvolvedores. As questões e as possíveis ideias de *design* sobre como modelar os itens de informação do domínio são extraídas automaticamente do metamodelo utilizado. A representação final do *design rationale* é feita com a linguagem OWL (*OWL Web Language*) [21], utilizada na Web Semântica. Isto permite o processamento computacional do *design rationale* para apoiar consultas sobre o *design*, a realização de inferências a partir do conhecimento registrado e o reúso dos modelos produzidos.

A arquitetura para utilização da abordagem Kuaba inclui três componentes principais: uma ferramenta de *design*, o subsistema Kuaba, e um framework de implementação de metamodelos [11]. A ferramenta de *design* é responsável por permitir a edição de modelos e pela notificação das modificações realizadas nos mesmos. O subsistema Kuaba então identifica o tipo de evento ocorrido na ferramenta e instancia os elementos corretos da ontologia Kuaba com as opções de *design* do metamodelo usadas no modelo, solicitando argumentos quando necessário. O framework de implementação de metamodelos fornece toda a infraestrutura para gerenciar a criação, o armazenamento, o acesso, a descoberta e as consultas de metadados baseados na especificação MOF (*Meta-Object Facility*) [14].

O subsistema Kuaba é o responsável pela implementação da abordagem Kuaba. Nele se encontram, entre outros, os módulos para a manipulação da ontologia Kuaba (*kuabaModel*), para captura e representação semiautomática de *design rationale* (*mofParser*) e para consulta e edição de *rationale* (*repository*). A arquitetura também inclui um módulo que deve ser implementado na ferramenta de *design* chamado *designToolAdapter*, que é responsável pelo recebimento dos eventos gerados em sua área de desenho. Isto torna o subsistema Kuaba independente de ferramenta de *design*. Essa arquitetura foi utilizada no desenvolvimento da ferramenta KSE (*Kuaba Software Engineering*) [11].

2.1 Ontologia Kuaba

A ontologia Kuaba define um conjunto de elementos (classes, propriedades, relações e restrições) para o domínio de *design rationale*. Nela, o *design rationale* é estruturado em três elementos de raciocínio principais: questões, que representam os problemas de *design*, ideias que representam possíveis soluções para esses problemas e argumentos que representam os prós e os contras dessas ideias. A ontologia também possui elementos

específicos para registrar as decisões tomadas, as justificativas para aceitação ou rejeição das possíveis ideias de solução, os artefatos gerados a partir das soluções aceitas e informações sobre a atividade de *design*, como o metamodelo utilizado e as pessoas envolvidas. A Figura 2 mostra um diagrama de classes UML (*Unified Modeling Language*) [15] para ilustrar os elementos de raciocínio da ontologia Kuaba utilizados neste trabalho.

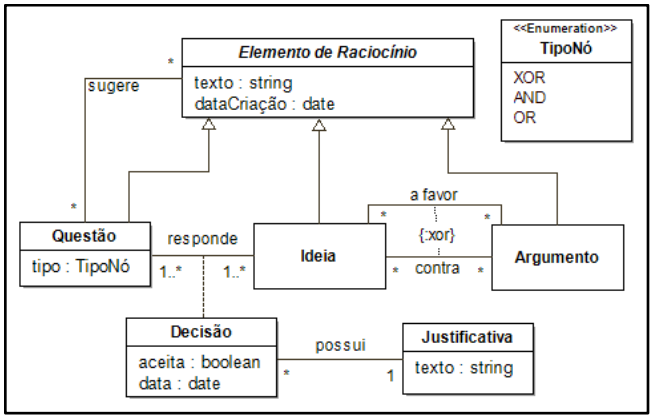

Figura 2. Ontologia Kuaba

Resumidamente, os elementos de raciocínio são divididos em três tipos: Questão, usado para representar problemas de *design* com os quais o desenvolvedor deve lidar; Ideia para representar as possíveis alternativas de solução para esses problemas; e Argumento que registra o conhecimento que os desenvolvedores empregaram no *design*. Um argumento pode ser a favor ou contra às alternativas apresentadas. Para a Questão pode ser definida a propriedade tipo de nó, que indica se as ideias de solução que respondem à questão devem ser consideradas mutuamente exclusivas ou não. A aceitação ou a rejeição de uma ideia como solução para uma questão é registrada pelo elemento Decisão que deve possuir uma justificativa.

2.2 *Design* Dirigido a Domínio (DDD)

DDD é uma abordagem para o desenvolvimento de sistemas de software complexos por meio do uso de padrões, divididos em estratégicos e táticos [4]. O ponto de partida do DDD é a construção de um modelo de domínio que deve ser construído em conjunto por especialistas do domínio e desenvolvedores de software, utilizando a linguagem ubíqua, que contém as expressões provenientes do domínio da aplicação [1].

Os padrões estratégicos descrevem práticas adequadas para aplicar quando há vários modelos de domínio e visam facilitar a evolução e coexistência desses diversos modelos. Já os padrões táticos, foco deste trabalho, são usados na identificação de classes baseadas em responsabilidades. Os padrões táticos considerados neste trabalho são: entidade, objeto valor, agregado e serviço.

Uma entidade é um objeto que representa um conceito do domínio com identidade única, independente dos valores de suas propriedades. Por exemplo, em um sistema de vendas, pode haver dois ou mais objetos que representam clientes. Mesmo que os valores da propriedade nome sejam iguais nesses objetos, eles representam indivíduos diferentes.

Um objeto valor é um objeto cujo estado não pode ser alterado após sua construção. Ele representa um aspecto descritivo do domínio que não possui uma identidade conceitual [4]. Sua identidade é baseada em seu estado (valores de seus atributos). Assim, dois objetos com o mesmo estado são considerados iguais. Um exemplo de conceito que pode ser modelado como objeto valor é endereço.

Um agregado é um conjunto de objetos associados tratado como uma unidade para fins de alterações de dados. Cada agregado tem uma raiz e um limite. O limite define o que está dentro do agregado. A raiz é uma única entidade específica contida no agregado, que os objetos externos são permitidos a manter referências, embora objetos dentro do limite possam conter referências uns aos outros [4]. Um exemplo de agregado no domínio de vendas é o conjunto de objetos formado por um pedido (entidade raiz) e seus itens, no qual o objeto pedido assume a responsabilidade de manter a integridade da coleção de itens de pedido associados.

Serviços representam operações que não pertencem conceitualmente a um objeto específico. Algumas delas são intrinsecamente atividades ou ações que podem envolver vários objetos [4]. Um exemplo de serviço é uma operação de validação de pedido na qual são verificados a disponibilidade dos produtos, o endereço de entrega e a confirmação do pagamento.

2.3 Perfil DDMM

O DDMM[1] (*Domain-driven Microservice Architecture Modeling* [17] é um perfil UML para apoiar o DDD de arquiteturas de microsserviços. Um perfil UML é uma extensão das metaclasses da linguagem UML por meio da definição de estereótipos e restrições. Assim, instâncias das metaclasses estendidas podem ser semanticamente enriquecidas com os estereótipos específicos do perfil.

O perfil DDMM fornece estereótipos para os principais padrões táticos e estratégicos do DDD, como mostra a Figura 3. Ele estende as metaclasses *Class*, *Operation*, *Package* e *Property* da linguagem UML. A relação entre um estereótipo e as metaclasses que ele estende é representada como uma seta com ponta de seta preenchida apontando do estereótipo para a metaclasse. Neste trabalho são utilizados apenas os estereótipos *Entity*, *ValueObject*, *Service*, *AggregatePart* e *AggregateRoot* que estendem a metaclasse *Class*.

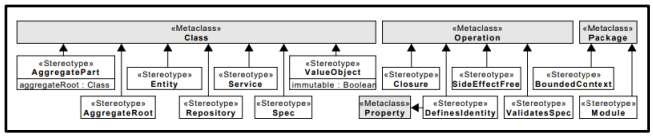

Figura 3. Estereótipos do perfil DDMM [17]

No sistema de vendas, citado como exemplo na seção 2.2, os estereótipos do perfil DDMM podem ser utilizados para identificar as classes dos objetos cliente, endereço, pedido, item de pedido e a operação de validação de pedido de acordo com o DDD. A Figura 4 mostra esse exemplo de uso do perfil DDMM.

A classe Cliente é marcada com o estereótipo *<<Entity>>* indicando que cada objeto dessa classe tem uma identidade única. A classe Endereço é marcada com o estereótipo *<<ValueObject>>*, indicando que um objeto dessa classe representa um aspecto descritivo do domínio. Pedido é marcado com os estereótipos *<<Entity>>* e *<<AggregateRoot>>* indicando que a classe é a raiz do agregado e ItemPedido é marcado com os estereótipos *<<Entity>>* e *<<AggregatePart>>*. Nessa classe há também a especificação da propriedade *aggregateRoot(Pedido)* do estereótipo *<<AggregatePart>>*

[1] https://github.com/SeelabFhdo/ddmm-uml-profile

indicando que um objeto ItemPedido só existe associado a um Pedido. Por fim, a classe ValidaçãoPedido é marcada com o estereótipo *<<Service>>* pois sua operação não pertence conceitualmente a um objeto específico.

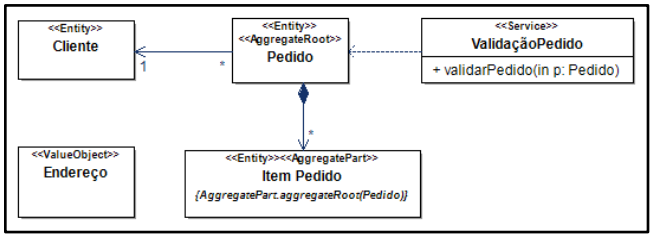

Figura 4. Exemplo de uso do perfil DDMM

2.4 MDE e a Arquitetura MDA

MDE é uma abordagem de desenvolvimento de software cujo foco são modelos ao invés de programas de computador. Essa abordagem tem como objetivo principal a geração automática de código a partir de modelos formalmente especificados. MDE permite que os modelos de software estejam muito mais próximos do domínio do problema do que das linguagens de programação. Assim, os modelos tornam-se menos sensíveis às tecnologias de computação, e com isso eles ficam mais fáceis de especificar, entender e manter [19]. Entre as arquiteturas propostas para apoiar a MDE destaca-se a MDA[13].

A arquitetura MDA permite especificar um sistema independentemente da plataforma técnica ou tecnologia utilizada. Uma especificação MDA geralmente consiste em três modelos: um modelo de domínio denominado CIM (*Computation Independent Model*), um modelo que define a funcionalidade e comportamento de negócio de maneira independente de tecnologia denominado PIM (*Platform Independent Model*), e um ou mais modelos específicos de plataforma, chamados PSM (*Platform Specific Model*).

Um CIM apenas descreve conceitos de negócio enquanto um PIM pode definir uma arquitetura de alto nível do sistema para atender as necessidades de negócios. Já um PSM é qualquer modelo que é mais específico de tecnologia que um PIM relacionado [13].

A base da MDA consiste nas transformações entre esses modelos de forma automática até a geração de código. Assim, é possível executar transformações no CIM para gerar o PIM, no PIM para gerar o PSM e, a partir do PSM gerar código. Essas transformações ocorrem a nível de metamodelo, mapeando elementos entre o metamodelo utilizado no modelo de entrada e o metamodelo do modelo alvo.

3. ABORDAGEM KUABA-MDE

Kuaba-MDE é uma abordagem de desenvolvimento de software baseada em DDD, que permite aos desenvolvedores elaborar modelos de discussão e conectar esses modelos aos demais artefatos de software por meio de transformações MDA. Ela é composta de: um modelo de discussão (CIM), construído utilizando os elementos da ontologia Kuaba; um modelo independente de plataforma (PIM), gerado via uma transformação de modelos, no qual os conceitos definidos no modelo de discussão são refinados pelo desenvolvedor de acordo com o domínio e os padrões táticos do DDD; e um modelo específico de plataforma (PSM), também gerado por regras de transformação definidas com base nesses padrões táticos. Kuaba-MDE também considera o uso de um mecanismo de geração automática de código a partir do PSM. A Figura 5 apresenta uma visão geral da abordagem.

O Modelo de Discussão (CIM) é construído pelo desenvolvedor durante sua interação com os especialistas de domínio e tem como objetivo registrar toda a deliberação sobre os conceitos fundamentais do domínio que serão considerados no *design*. O uso dos elementos de raciocínio da ontologia Kuaba na representação desse modelo permite lidar com a incerteza no *design*, uma vez que todos os conceitos discutidos ficam registrados, assim como os argumentos (opiniões) e as decisões tomadas sobre tratá-los ou não no *design* do software. O modelo de domínio, que será a base do PIM, será gerado a partir dessas decisões.

No DDD, o modelo de domínio deve ser construído em conjunto por especialistas do domínio e desenvolvedores de software com a utilização de uma linguagem ubíqua, que tem como propósito facilitar a comunicação entre eles [1]. Como essa linguagem deve conter expressões provenientes do domínio da aplicação, o Modelo de Discussão contribui para a definição dessas expressões e, assim, ajuda a registrar o processo de criação da linguagem ubíqua utilizada durante o *design*.

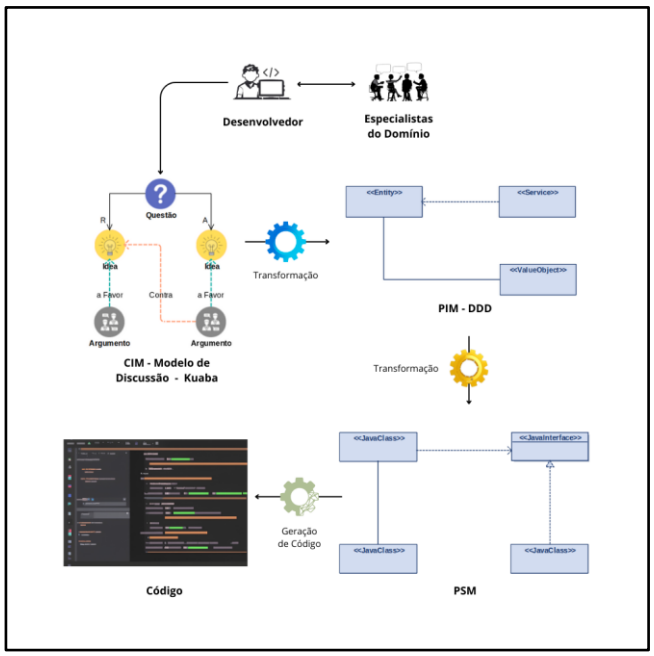

Figura 5. Abordagem Kuaba-MDE

O PIM é um modelo de classes gerado com base nos conceitos definidos no Modelo de Discussão. No PIM o desenvolvedor definirá as responsabilidades de cada classe de acordo com os padrões táticos do DDD (*Entity*, *ValueObject*, *Aggregate* e *Service*). Para isto, são utilizados os estereótipos do perfil DDMM apresentado na seção 2.3.

O PSM também é um modelo de classes gerado a partir do PIM por meio da execução de regras de transformação definidas com base nos estereótipos do perfil DDMM. Os elementos desse modelo são gerados com estereótipos definidos de acordo com o metamodelo da linguagem de programação que será utilizada na geração automática de código. Neste trabalho, a geração de código é feita na linguagem Java.

4. SUPORTE PARA KUABA-MDE

Para a implementação da abordagem proposta foi selecionada a ferramenta de modelagem Modelio[2]. Os critérios de seleção consideraram o fato dessa ferramenta possuir código aberto, ter documentação atualizada disponível, além de fornecer API (*Application Programming Interface*) e suporte nativo para MDA.

Modelio é um ambiente de modelagem de código aberto, baseado na arquitetura Eclipse, com suporte para as linguagens UML e BPMN (*Business Process Model and Notation*) [12], entre outras. A ferramenta possui suporte à MDE por meio de seu mecanismo de extensão.

A extensão da ferramenta Modelio é realizada por meio da criação de módulos. Os módulos são *plugins* desenvolvidos e compilados dentro do ambiente Eclipse e se comunicam com a ferramenta através da API fornecida. Esses módulos contêm todos os artefatos necessários para a extensão, incluindo código, perfis, ícones e documentação. A flexibilidade dos módulos permite que eles sejam adicionados, retirados ou combinados com outros módulos. Os módulos são compilados em arquivos .jmdac que podem ser adicionados diretamente ao catálogo de módulos da ferramenta e carregados de acordo com a necessidade de cada projeto. A abordagem proposta neste trabalho foi implementada por meio do desenvolvimento do módulo Kuaba[3] e da utilização do módulo *Java Designer*[4] para geração automática de código em linguagem Java.

4.1 Módulo Kuaba

O módulo Kuaba contém recursos para apoiar a criação e edição de modelos (CIM e PIM), a execução de transformações de modelos (de CIM para PIM e PIM para PSM), e geração automática de código em Java a partir do modelo PSM.

A implementação do módulo foi realizada a partir de um *template* disponibilizado pela própria Modelio. Esse *template* fornece uma estrutura base para o módulo e as bibliotecas necessárias, que possibilitam o desenvolvimento da interface gráfica e das novas funcionalidades.

Para dar suporte à criação de modelos de discussão, à execução das transformações MDA e à geração de código a partir do PSM foi definido o perfil Kuaba e implementados novos recursos na interface gráfica da ferramenta Modelio. Esses recursos incluem: novas opções de comandos no menu contextual; uma nova aba de propriedades Kuaba; e um guia prático para guiar o usuário da instalação do módulo Kuaba à criação dos modelos e execução das transformações.

4.1.1 Perfil Kuaba

O perfil Kuaba define um conjunto de estereótipos e ícones para permitir a criação de um diagrama específico para a representação gráfica de modelos de discussão (CIM) de acordo com a ontologia Kuaba. A Figura 6 mostra os elementos desse perfil. O perfil Kuaba estende as metaclasses *Class*, *Association, Dependency* e *Static Diagram* da UML com estereótipos específicos para representação dos elementos de raciocínio (*Question*, *Idea* e *Argument*) da ontologia Kuaba, suas propriedades e as relações entre eles.

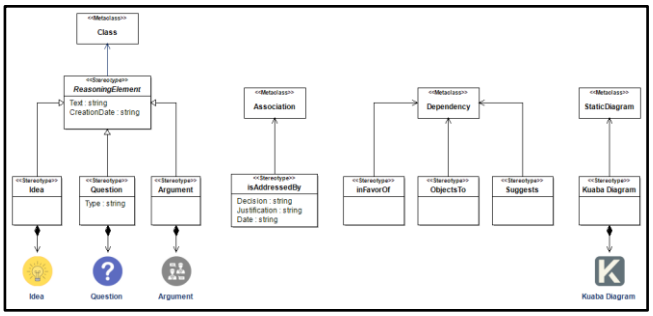

Figura 6. Perfil Kuaba

Os estereótipos dos elementos de raciocínio que estendem a metaclasse *Class* possuem as propriedades *Text* e *CreationDate*. O estereótipo *Question* possui também a propriedade *Type* cujo valor pode ser AND, OR ou XOR para indicar se a aceitação de ideias para a questão deve ser mutuamente exclusiva ou não, de acordo com a ontologia Kuaba. Já os estereótipos *Argument* e *Idea* possuem apenas as propriedades herdadas do estereótipo *ReasoningElement*.

Os estereótipos *IsAddressedBy*, *InFavorOf*, *ObjectsTo* e *Suggests* criados para marcar as relações entre os elementos de raciocínio estendem as metaclasses *Association* e *Dependency*. Esses estereótipos foram definidos respeitando os escopos e as propriedades presentes na ontologia Kuaba. O estereótipo *IsAddressedBy* pode ser utilizado em uma associação entre os elementos *Question* e *Idea*. Ele indica que uma questão é respondida por uma ideia. Para esse estereótipo são definidas as propriedades *Decision*, *Justification* e *Date*. A propriedade *Decision* pode receber os valores *Accepted* ou *Rejected*, representando que a ideia foi aceita ou rejeitada como uma solução para a questão que ela responde. A propriedade *Justification* registra o texto da justificativa para a decisão tomada. Já a propriedade *Date* indica a data em que a decisão foi tomada. Os estereótipos *InFavorOf* e *ObjectsTo* podem ser usados em relações de dependência entre os elementos *Argument* e *Idea*, mostrando que um argumento pode ser a favor ou contra uma ou mais ideias. Por fim, o estereótipo *Suggests* pode ser utilizado em uma dependência entre uma questão e qualquer outro elemento de raciocínio para representar que a partir de uma questão, ideia ou argumento pode surgir uma nova questão a ser resolvida.

4.1.2 Interface Gráfica do Módulo Kuaba

Na implementação da interface gráfica foi criada uma estrutura de pacotes específica para o módulo Kuaba na área *model explorer* da ferramenta Modelio, visando organizar os modelos previstos e seus respectivos diagramas, como ilustrado na Figura 7. Os pacotes criados são *Kuaba Package*, *CIM Package*, *PIM Package* e *PSM Package*. Estes pacotes possuem escopos bem definidos. Todos os elementos e funcionalidades do módulo, só podem ser utilizados dentro de um *Kuaba Package*, que é o pacote raiz do módulo. Já os outros pacotes (CIM, PIM e PSM) serão utilizados dentro desse pacote raiz e são responsáveis por determinar os escopos dos elementos, diagramas e transformações MDA relacionados a eles.

Para apoiar a criação dos elementos dos modelos previstos na abordagem Kuaba-MDE foram desenvolvidas novas opções para o menu contextual da Modelio, inserindo novos comandos. Os

[2] https://www.modelio.org/index.htm

[3] https://github.com/Autor/Kuaba-Software-Engineering-tool/tree/master/Kuaba_Module

[4] https://www.modeliosoft.com/en/modules/modelio-java-designer.html

comandos do menu contextual são elementos internos disponibilizados pela própria ferramenta em um arquivo de configuração. Esses elementos são responsáveis por invocar os métodos das classes configuradas para cada um deles. Existem classes padrão/pré-definidas para a criação de elementos, porém torna-se necessário criar novas classes para maior liberdade e estilização. Para a criação dos pacotes foi utilizada a classe *ElementCreate*, responsável pela criação de elementos. Na instanciação dessa classe devem ser informados como parâmetros o nome, a metaclasse e o estereótipo do novo elemento.

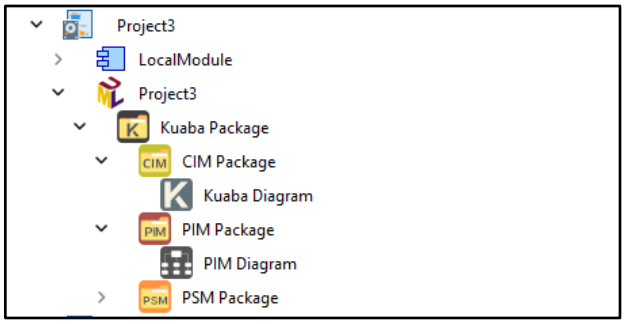

Figura 7. Estrutura de pacotes do Módulo Kuaba

A criação dos diagramas pode ser realizada de duas formas: pelo menu contextual ou pelo *wizard* da Modelio. A criação do diagrama pelo menu contextual foi elaborada da mesma forma que os pacotes, configurando um comando e utilizando a classe padrão *DiagramCreate*.

O *wizard* é a tela aberta a partir da opção *Create diagram* no menu contextual já disponível na ferramenta Modelio. Esta tela exibe opções para criação de todos os diagramas, independente do módulo em que estejam, porém respeitando o escopo de cada módulo. Nela, o usuário seleciona o diagrama e pode visualizar informações como nome, pacote no qual o diagrama será criado, uma imagem de exemplo e uma descrição, como exibido na Figura 8.

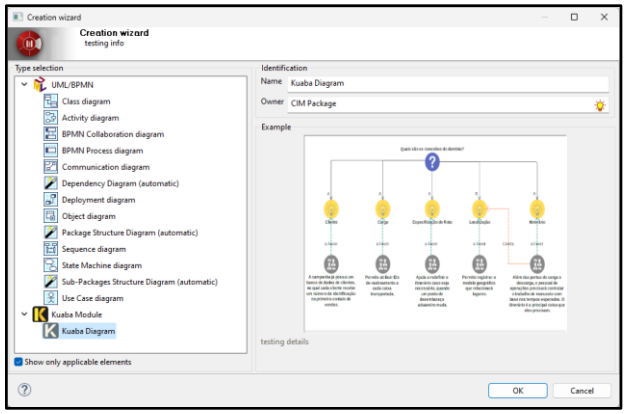

Figura 8. *Wizard* para a criação de diagramas

Para a adição do diagrama Kuaba no *wizard* foi necessário implementar novas classes. Essas classes foram divididas em dois segmentos, classes para apoiar a criação de diagramas e classes para o wizard. As classes para criação de diagramas são utilizadas tanto para o wizard de criação como para as transformações. A Figura 9 mostra o diagrama com as classes definidas. Os parâmetros e retornos das operações foram suprimidos para melhorar a legibilidade do diagrama.

A classe *DiagramBase* é a base para os diagramas. Ela possui um atributo protegido do tipo *AbstractDiagram*, que registra o diagrama que será criado. Isto é necessário para inserir modificações de estilo e para manipular, incluir ou remover elementos do diagrama. A classe possui também o método *getElement* utilizado para recuperar o valor desse atributo. O método construtor da classe recebe as informações relacionadas ao diagrama, verifica o tipo do diagrama, o cria e inclui a configuração de estilo. Esta classe é estendida pelas classes *KuabaDiagram* e *PIMDiagram* que apenas executam o método construtor herdado passando as informações específicas dos mesmos.

No segundo segmento, classes para o *wizard*, foram implementadas duas classes, *KuabaDiagramWizardContributor* e *PIMDiagramWizardContributo*r. Ambas as classes possuem uma estrutura bastante semelhante e estendem a mesma classe *AbstractDiagramWizardContributor* da própria Modelio. Os métodos dessas classes sobrescrevem os métodos herdados. O método *actionPerformed* é responsável pela criação do diagrama com os atributos informados. O método *getCreatedElementType,* retorna um *ElementDescriptor*, representando o tipo do elemento, recebendo sua metaclasse e o estereótipo. Por fim, o método *checkCanCreateIn* verifica se é possível criar o diagrama no pacote desejado.

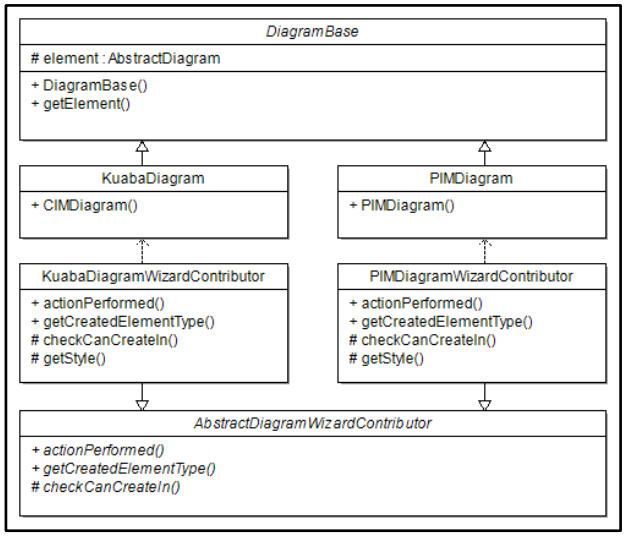

Figura 9. Hierarquia das classes do *wizard* de criação

Além das novas opções no menu contextual e do wizard, também foi desenvolvida a aba de propriedades do módulo Kuaba. Esta aba permite a edição de todas as propriedades dos elementos do modelo de discussão (diagrama Kuaba), conforme as definições da ontologia Kuaba, em uma área exclusiva, melhorando a experiência do usuário.

Para a implementação da aba foram criadas novas classes tanto para lidar com a aba em si como para lidar com as propriedades dos elementos. A Figura 10 ilustra as classes criadas. Os parâmetros e retornos das operações foram suprimidos para melhorar a legibilidade do diagrama.

Para a aba, foi definida a classe *KuabaPropertyPage*. Ela estende a classe *AbstractModulePropertyPage*, sobrescrevendo os métodos *changeProperty* e *update*. O primeiro é encarregado de verificar se o valor de uma propriedade foi alterado na aba. Para isto, ele recupera o elemento que foi selecionado pelo usuário e a propriedade da aba que foi alterada, passando o novo valor. O método *update* lida com a exibição da aba, ou seja, quando o

usuário seleciona outro elemento no diagrama, o método recupera as propriedades do novo elemento selecionado e as exibe na aba.

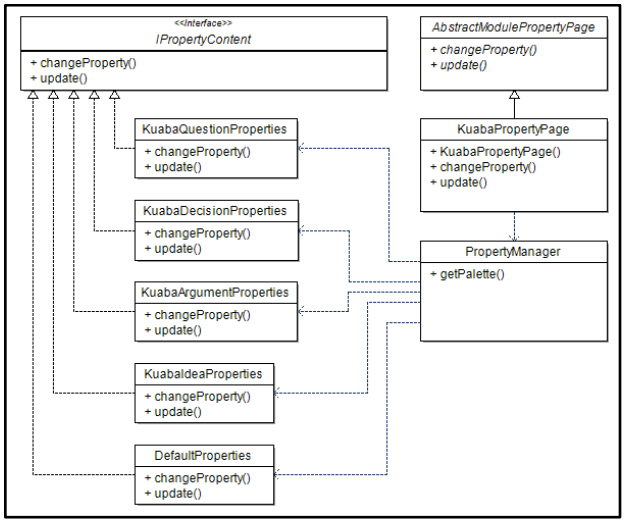

Figura 10. Classes para aba de propriedades Kuaba

Para o gerenciamento das propriedades dos elementos do modelo de discussão na aba, também foram definidas novas classes e uma interface. A interface IPropertyContent também possui as operações changeProperty e update , citadas anteriormente. No entanto, eles recebem como parâmetro um único elemento, não uma lista. As classes definidas para as propriedades da aba dos elementos da ontologia Kuaba implementam essa interface.

Nas classes, o método changeProperty verifica qual propriedade do elemento foi modificada na aba e altera o valor da propriedade do estereótipo em si. Já o método update atualiza as propriedades da aba com o valor definido nas propriedades do estereótipo no momento em que são exibidas na mesma. A classe de gerenciamento PropertyManager apenas busca o elemento selecionado, verifica seu estereótipo e retorna uma nova instância da classe de aba relacionada ao elemento.

4.1.3 Transformações de Modelos

As transformações em MDA ocorrem a nível de metamodelo, mapeando elementos entre o metamodelo usado no modelo de origem e o metamodelo do modelo destino, de forma automática [13]. Esse mapeamento ocorre com o auxílio de marcações (estereótipos) usadas para marcar os elementos do modelo e orientar a transformação.

Neste trabalho foram implementadas duas transformações: CIM para PIM e PIM para PSM. Para a execução dessas transformações foram incluídos dois comandos específicos no menu contextual do módulo Kuaba: *Transform CIM to PIM*, executado a partir do pacote CIM; e *Transform PIM to PSM*, executado a partir do pacote PIM. A Figura 11 ilustra o comando *Transform CIM to PIM* no menu contextual a partir do pacote CIM.

As transformações entre os modelos são realizadas por meio das funções de mapeamento implementadas nas classes ilustradas na Figura 12. Os parâmetros e retornos das operações foram suprimidos para melhorar a legibilidade do diagrama.

As classes responsáveis pelas transformações são *TransformCIMToPIM* e *TransformPIMToPSM*. Essas classes estendem a classe abstrata *DefaultModuleCommandHandler* disponível na API da ferramenta Modelio. Os métodos *accept* e *actionPerformed* são sobrescritos nas classes responsáveis pelas transformações.

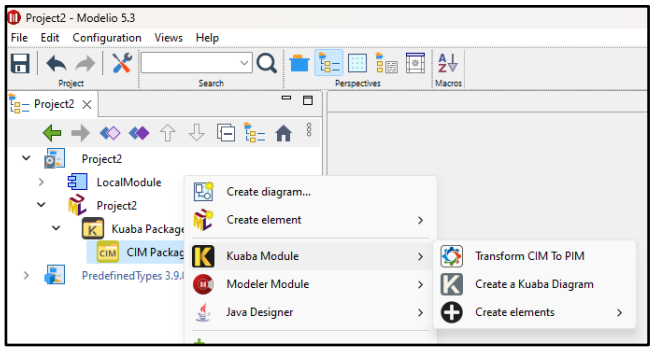

Figura 11. Comando Transform CIM To PIM

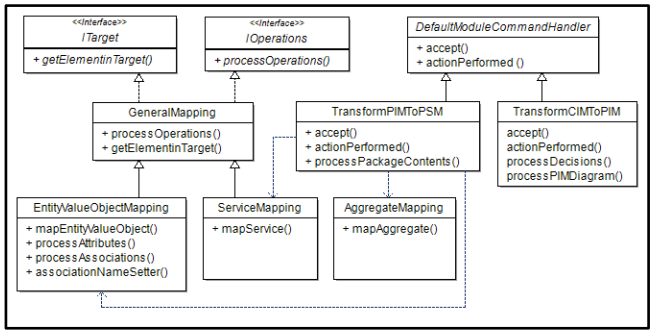

Figura 12. Classes para Transformações entre Modelos

O método *accept* tem a função de verificar o escopo, ou seja, verifica se a opção de transformação é exibida para o elemento selecionado pelo usuário, permitindo ou não a transformação. Se a transformação for permitida, o método *actionPerformed* é executado. A Figura 13 mostra o código do método *actionPerformed* na classe *TransformCIMToPIM*.

O método *actionPerformed* garante que o elemento selecionado pelo usuário é um pacote e recupera a sessão atual do projeto (linhas 3-7). Na classe *TransformCIMToPIM*, este método recupera os elementos do pacote CIM marcados com o estereótipo *Question* (linhas 9-12), executa o método *processDecisions* para cada elemento recuperado (linhas 21-24) e, por fim, executa o método *processPIMDiagram* (linha 27).

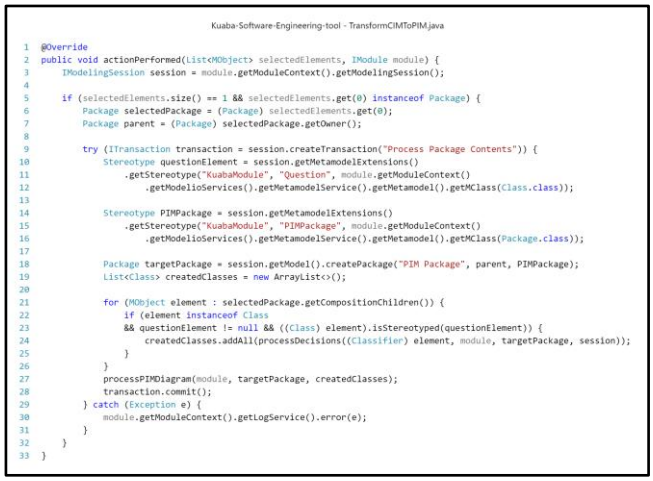

Figura 13. Método *actionPerformed* em *TransformCIMToPIM*

O método *processDecisions*, exibido na Figura 14, recupera todas as associações referentes ao elemento *Question* (linha 3), que são associações marcadas com o estereótipo *IsAddressedBy* e, para

cada uma, verifica se a propriedade *Decision* contém o valor *Accepted* (linhas 7 - 9). Se for o caso, uma nova classe é criada com o mesmo nome do elemento marcado com o estereótipo *Idea* (linha 10), vinculado à associação. Já o método *processPIMDiagram*, cria o diagrama PIM e insere as classes criadas no diagrama, de forma organizada.

Kuaba-Software-Engineering-tool - TransformCIMToPIM.java

```
1  private List<Class> processDecisions(Classifier element, IModule module, Package targetPackage, IModelingSession session) {
2
3      EList<AssociationEnd> associationEnds = element.getOwnedEnd();
4      List<Class> createdClasses = new ArrayList<>();
5
6      try (ITransaction t = session.createTransaction("Process Package Contents")) {
7          for (AssociationEnd associationEnd : associationEnds) {
8
9              if ("A".equals(associationEnd.getName())) {
10                 Class newClass = session.getModel().createClass(associationEnd.getTarget().getName(), targetPackage, null);
11                 createdClasses.add(newClass);
12             }
13         }
14
15         t.commit();
16     } catch (Exception e) {
17         module.getModuleContext().getLogService().error(e);
18     }
19     return createdClasses;
20 }
```

Figura 14. Método *processDecisions*

Na classe *TransformPIMToPSM*, o método *actionPerformed*, além de garantir que o elemento selecionado pelo usuário é um pacote, ele acessa a sessão atual do projeto, cria o pacote PSM e executa o método *processPackageContents*, responsável pela execução das funções de mapeamento para transformação do PIM para PSM.

O método *processPackageContents* é exibibo na Figura 15. Ele recupera alguns dos estereótipos do perfil DDMM importados no projeto (linhas 4 - 15) e percorre as classes do pacote PIM (linhas 17 - 18), verificando se estão marcadas com estes estereótipos. Se estiverem marcadas, o método executa as funções de mapeamento correspondentes para gerar os elementos do pacote PSM de acordo com esses estereótipos (linhas 19 - 30).

As funções de mapeamento são implementadas pelas classes *EntityValueObjectMapping*, *AggregateMapping* e *ServiceMapping*. Para evitar duplicação de código foi criada a classe *GeneralMapping*, que implementa as interfaces *IOperations* e *ITarget* para tratar as operações definidas nas classes do PIM e copiá-las para as classes geradas no PSM, como mostra a Figura 12.

A interface *ITarget* recupera a referência da classe já criada no pacote PSM através do nome, permitindo realizar alterações nas funções de mapeamento descritas a seguir.

4.1.3.1 Mapeamento de Entidades e Objetos Valor

As classes do pacote PIM marcadas com os estereótipos *Entity* e *ValueObject* são mapeadas para classes Java no pacote PSM. Esse mapeamento é realizado pelo método *mapEntityValueObject* da classe *EntityValueObjectMapping*, ilustrado na Figura 16.

No início da execução, uma referência ao estereótipo *JavaClass* do módulo *JavaDesigner* é recuperada (linha 3 - 5). Isso permite a geração de código Java para as classes marcadas com esse estereótipo no PSM. Com a referência do estereótipo adquirida, é criada uma classe no pacote de destino (PSM) com o mesmo nome da classe do PIM (linha 9) que está sendo atualmente processada na classe principal.

Para cada classe marcada com o estereótipo *Entity* ou *ValueObject* são executados quatro métodos: *processAttributes* para copiar atributos (linha 10), *processOperations* para criar operações (linha 11), *processAssociations* para criar associações (linha 12), e *associationNameSetter* método auxiliar para declarar o nome da associação (linha 13), pois isso não é feito automaticamente pela ferramenta Modelio. A Figura 17 mostra o código do método *processAttributes*.

Kuaba-Software-Engineering-tool - TransformPIMToPSM.java

```
1  public void processPackageContents(IModelingSession session, Package sourcePackage, IModule module, Package target) {
2
3      try (ITransaction transaction = session.createTransaction("Process Package Contents")) {
4          Stereotype entityStereotype = session.getMetamodelExtensions()
5              .getStereotype("LocalModule", "Entity", module.getModuleContext().getModelioServices()
6                  .getMetamodelService().getMetamodel().getMClass(Class.class));
7          Stereotype valueObjectStereotype = session.getMetamodelExtensions()
8              .getStereotype("LocalModule", "ValueObject", module.getModuleContext().getModelioServices()
9                  .getMetamodelService().getMetamodel().getMClass(Class.class));
10         Stereotype serviceStereotype = session.getMetamodelExtensions()
11             .getStereotype("LocalModule", "Service", module.getModuleContext().getModelioServices()
12                 .getMetamodelService().getMetamodel().getMClass(Class.class));
13         Stereotype aggregateRootStereotype = session.getMetamodelExtensions()
14             .getStereotype("LocalModule", "AggregateRoot", module.getModuleContext().getModelioServices()
15                 .getMetamodelService().getMetamodel().getMClass(Class.class));
16
17         for (MObject element : sourcePackage.getCompositionChildren()) {
18             if (element instanceof Class) {
19                 if (entityStereotype != null && ((Class) element).isStereotyped(entityStereotype)
20                 || valueObjectStereotype != null && ((Class) element).isStereotyped(valueObjectStereotype)) {
21                     EntityValueObjectMapping mapping = new EntityValueObjectMapping();
22                     mapping.mapEntityValueObject(session, module, target, (Class) element);
23                 }
24                 if (serviceStereotype != null && ((Class) element).isStereotyped(serviceStereotype)) {
25                     ServiceMapping mp = new ServiceMapping();
26                     mp.mapService(session, module, target, (Class) element);
27                 }
28                 if (aggregateRootStereotype != null && ((Class) element).isStereotyped(aggregateRootStereotype)) {
29                     AggregateMapping mp = new AggregateMapping();
30                     mp.mapAggregate(session, module, target, (Class) element);
31                 }
32             }
33         }
34         transaction.commit();
35     } catch (Exception e) {
36         module.getModuleContext().getLogService().error(e);
37     }
38 }
```

Figura 15. Método *processPackageContents*

Kuaba-Software-Engineering-tool - EntityValueObjectMapping.java

```
1  public void mapEntityValueObject(IModelingSession session, IModule module, Package target, Class element) {
2
3      Stereotype javaClassStereotype = session.getMetamodelExtensions()
4          .getStereotype("JavaDesigner", "JavaClass", module.getModuleContext().getModelioServices()
5              .getMetamodelService().getMetamodel().getMClass(Class.class));
6      try (ITransaction transaction = session.createTransaction("Process Entity")) {
7
8          System.out.println("Entered transaction for Entity");
9          Class myClass = session.getModel().createClass(element.getName(), target, javaClassStereotype);
10         processAttributes(session, module, myClass, element);
11         processOperations(session, module, myClass, element);
12         processAssociations(myClass, (Classifier) element, module);
13         associationNameSetter(module, session, target, element);
14
15         transaction.commit();
16     } catch (Exception e) {
17         module.getModuleContext().getLogService().error(e);
18     }
19 }
```

Figura 16. Método *mapEntityValueObject*

Kuaba-Software-Engineering-tool - EntityValueObjectMapping.java

```
1  public void processAttributes(IModelingSession session, IModule module, Class targetClass, MObject sourceElement) {
2
3      try (ITransaction transaction = session.createTransaction("Process Attributes")) {
4          for (Attribute sourceAttribute : ((Classifier) sourceElement).getOwnedAttribute()) {
5              session.getModel().createAttribute(sourceAttribute.getName(), sourceAttribute.getType(), targetClass);
6          }
7          transaction.commit();
8      }  catch (Exception e) {
9          module.getModuleContext().getLogService().error(e);
10     }
11 }
```

Figura 17. Método *processAttributes*

O método *processAttributes* percorre cada atributo da classe que está sendo processada no PIM e cria um atributo correspondente na nova classe gerada no PSM (linhas 4 - 5). Um comportamento semelhante é realizado para as operações de cada classe no método *processOperations*. Com as classes do PIM mapeadas para classes no PSM com seus atributos e operações, é executado o método *processAssociations*, ilustrado na Figura 18.

Kuaba-Software-Engineering-tool - EntityValueObjectMapping.java

```
1  public void processAssociations(Class myClass, Classifier element, IModule module) {
2
3      EList<AssociationEnd> associationEnds = element.getOwnedEnd();
4      List<MObject> associationList = new ArrayList<MObject>();
5
6      for (AssociationEnd associationEnd : associationEnds) {
7          associationList.add((MObject) associationEnd);
8      }
9
10     IModelManipulationService manipulationService = module.getModuleContext()
11         .getModelioServices().getModelManipulationService();
12     manipulationService.copyTo(associationList, (MObject) myClass);
13 }
```

Figura 18. Método *processAssociations*

Em um diagrama de classes UML, uma associação é um relacionamento representado graficamente por uma linha, cujas extremidades (a*ssociationEnd*) estão conectadas a classes. Assim, para copiar as associações entre classes do PIM para o PSM, o método *processAssociations* recupera (linha 3) e processa também as extremidades de cada associação identificada no PIM (linha 6 - 7) e as copia para o PSM (linha 12).

Além de copiar as associações, também é necessário copiar seus nomes para o PSM. Isto é feito pelo método auxiliar *associationNameSetter*. Esse método auxiliar foi desenvolvido para contornar uma limitação presente na própria ferramenta Modelio, na qual as associações criadas não possuem um nome, apenas um tipo. Para resolver esse problema, uma lista de extremidades das associações definidas para cada classe gerada no PSM é recuperada e um nome é criado para a associação que foi copiada do PIM.

4.1.3.2 Mapeamento de Serviços

As classes do pacote PIM marcadas com o estereótipo *Service* também são mapeadas para classes Java no pacote PSM. No entanto, como um serviço no DDD define um objeto sem estado criado para implementar operações que não são responsabilidades de uma entidade ou objeto valor específico, esse mapeamento também inclui a definição de interfaces Java no PSM. O mapeamento de serviços é realizado pelo método *mapService* da classe *ServiceMapping*. Esse método é ilustrado na Figura 19.

O método *mapService* cria uma classe com o estereótipo *JavaClass* (linha 10) e uma interface com o estereótipo *JavaInterface* (linhas 11 - 12) dentro do pacote PSM e estabelece uma relação de implementação entre a classe e a interface (linha 13). Também executa a operação *processOperations* para copiar as operações da classe marcada com o estereótipo *Service* para a classe e para a interface criadas no PSM (linhas 18 - 19). Na interface essas operações são definidas como abstratas.

Kuaba-Software-Engineering-tool - ServiceMapping.java

```
1  public void mapService(IModelingSession session, IModule module, Package target, Class element) {
2      Stereotype javaClassStereotype = session.getMetamodelExtensions()
3          .getStereotype("JavaDesigner", "JavaClass", module.getModuleContext().getModelioServices()
4              .getMetamodelService().getMetamodel().getMClass(Class.class));
5      Stereotype javaInterfaceStereotype = session.getMetamodelExtensions()
6          .getStereotype("JavaDesigner", "JavaInterface", module.getModuleContext().getModelioServices()
7              .getMetamodelService().getMetamodel().getMClass(Interface.class));
8      try (ITransaction transaction = session.createTransaction("Process Service")) {
9
10         Class myClass = session.getModel().createClass(element.getName(), target, javaClassStereotype);
11         Interface myInterface = session.getModel()
12             .createInterface("I" + element.getName(), target, javaInterfaceStereotype);
13         InterfaceRealization realization = session.getModel().createInterfaceRealization();
14
15         realization.setImplementer(myClass);
16         realization.setImplemented(myInterface);
17
18         processOperations(session, module, myClass, element);
19         processOperations(session, module, myInterface, element);
20
21         for (Operation operation : myInterface.getOwnedOperation()) {
22             operation.setIsAbstract(true);
23         }
24         transaction.commit();
25     } catch (Exception e) {
26         module.getModuleContext().getLogService().error(e);
27     }
28 }
```

Figura 19. Método *mapService*

4.1.3.3 Mapeamento de Agregados

O mapeamento de agregados requer o processamento das classes do PIM marcadas com os estereótipos *AggregateRoo*t e *AggregatePart* e das associações entre essas classes. A Figura 20 mostra o método *mapAggregate* responsável por esse processamento.

O método *mapAggregate* é executado quando uma classe é marcada com o estereótipo *AggregateRoot*, pois a lógica desse mapeamento inicia-se a partir da raiz do agregado. Inicialmente, é obtida uma referência do estereótipo *AggregatePart* do perfil DDMM (linhas 3 - 5), importado para o módulo local da ferramenta. Em seguida, é recuperada uma lista com as extremidades das associações (linhas 9 - 10) vinculadas à classe marcada com o estereótipo *AggregateRoot*, processadas anteriormente, uma vez que a raiz de um agregado no DDD deve ser uma entidade. A partir dessa lista, uma iteração verifica se o elemento oposto a cada extremidade está estereotipado como *AggregatePart* (linhas 12 - 13). Se estiver, a extremidade vinculada à classe marcada como *AggregateRoot* é alterada para uma composição no pacote PSM (linha 14). Essa alteração é necessária porque a existência da parte de um agregado está vinculada à existência da raiz do agregado. Portanto, se a raiz for removida, as partes também devem ser removidas.

Kuaba-Software-Engineering-tool - AggregateMapping.java

```
1  public void mapAggregate(IModelingSession session, IModule module, Package target, Class element) {
2
3      Stereotype aggregatePartStereotype = session.getMetamodelExtensions()
4          .getStereotype("LocalModule", "AggregatePart", module.getModuleContext().getModelioServices()
5              .getMetamodelService().getMetamodel().getMClass(Class.class));
6
7      try (ITransaction transaction = session.createTransaction("Process Aggregate")) {
8
9          EList<AssociationEnd> associationEnds = getElementinTarget(session, module, target, element)
10             .getOwnedEnd();
11
12         for (AssociationEnd associationEnd : associationEnds) {
13             if (associationEnd.getOpposite().getOwner().isStereotyped(aggregatePartStereotype)) {
14                 associationEnd.setAggregation(AggregationKind.KINDISCOMPOSITION);
15             }
16         }
17         transaction.commit();
18     } catch (Exception e) {
19         module.getModuleContext().getLogService().error(e);
20     }
21 }
```

Figura 20. Método *mapAggregate*

4.1.4 Geração automática de Código

A ferramenta Modelio possui recursos para geração automática de código a partir de modelos UML para linguagens como Java, C++, C# e SQL. Neste trabalho utilizamos o módulo *Java Designer* que gera código fonte e anotações na linguagem Java. Os códigos gerados são gravados na pasta "src" do diretório do projeto. O código é gerado a partir das classes marcadas com os estereótipos disponíveis no módulo *Java Designer* como *JavaClass* e *JavaInterface*, entre outros.

5. EXEMPLO DE USO

Para utilizar o módulo Kuaba, o arquivo *KuabaModule.jmdac*[5] deve ser importado para o catálogo de módulos da ferramenta Modelio. Para isto, é necessário criar um projeto, adicionar o arquivo ao catálogo e implantar o módulo Kuaba no projeto utilizando as opções *Modules catalog* e *Modules* do item de menu *Configuration*. Após essa configuração inicial, o módulo Kuaba está disponível para a criação do pacote *Kuaba package*, como mostra a Figura 21. Nesse pacote poderão ser gerenciados os modelos e executadas as transformações entre eles.

Além da implantação do módulo Kuaba, é necessário importar o perfil DDMM no módulo local do projeto para que os estereótipos definidos para os padrões táticos do DDD possam ser utilizados na criação do modelo de *design* (PIM). A importação do perfil pode ser realizada clicando com botão direito do mouse no *LocalModule* do projeto e selecionando as opções *Import/XMI profile import*.

Neste exemplo de uso é utilizada uma versão simplificada do domínio de transporte de cargas, definida a partir dos exemplos apresentados em [4]. Nesse domínio, um cliente pode contratar o transporte de diversas cargas. Para cada carga é definida uma

[5] https://github.com/Autor/Kuaba-Software-Engineering-tool/releases

especificação de rota e um itinerário. A especificação da rota indica os requisitos da entrega e o itinerário contém o plano de entrega operacional. O itinerário deve seguir a especificação da rota. Por isso, quando o ponto de desembaraço aduaneiro muda é necessário alterar todo o plano da rota.

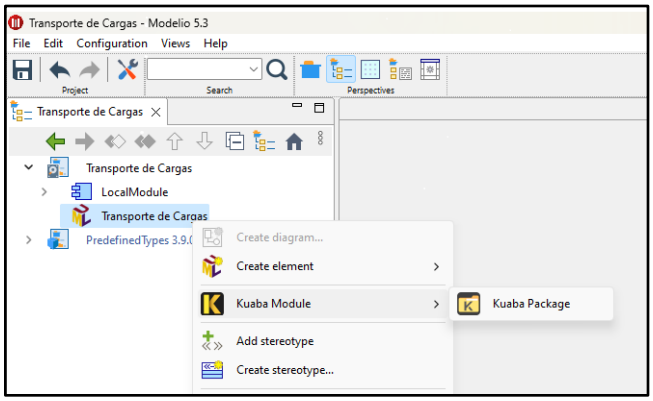

Figura 21. Módulo Kuaba

5.1 Criação do Modelo de Discussão (CIM)

Na abordagem Kuaba-MDE, inicialmente o desenvolvedor cria o modelo de discussão. Para isto, ele deve criar um pacote CIM clicando com o botão direito no *Kuaba package* e selecionando as opções *Kuaba Module/CIM Package*. Após criar o pacote CIM, é possível criar um diagrama Kuaba, clicando com o botão direito sobre esse pacote e selecionando as opções *Kuaba Module/Create Kuaba diagram.*

A Figura 22 ilustra o modelo de discussão resultante da deliberação entre um desenvolvedor de software e os especialistas de domínio sobre os conceitos relevantes para o software de transporte de cargas.

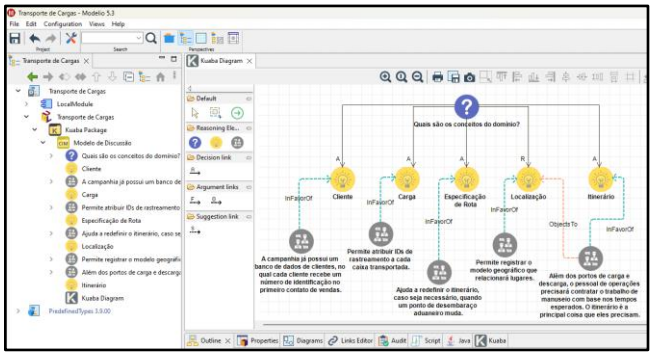

Figura 22. CIM para o domínio de Transporte de Cargas

No modelo de discussão os conceitos do domínio são representados como ideias (ícones de lâmpadas) que respondem à questão principal (ícone de interrogação). Nesse exemplo, os conceitos considerados são Cliente, Carga, Especificação de Rota, Localização e Itinerário. Para cada conceito são registrados os argumentos contra (seta em vermelho - *objectsTo*) e a favor (setas em verde - *InFavorOf*) e as decisões tomadas sobre aceitar (A) ou rejeitar (R) o conceito para a questão principal sobre o domínio. É possível observar que o conceito de Localização foi rejeitado e substituído pelo conceito Itinerário.

5.2 Criação do Modelo de *Design* com DDD

Um modelo de *design* (PIM) inicial pode ser obtido a partir do modelo de discussão por meio de uma transformação CIM para PIM. Para executar essa transformação, o desenvolvedor deve clicar com o botão direito do mouse sobre o pacote CIM e selecionar as opções *Kuaba Module/Transform CIM to PIM*. Como resultado, um pacote PIM e um diagrama de classes com os conceitos aceitos no modelo de discussão são automaticamente criados, como mostra a Figura 23. Esse diagrama pode, então, ser editado pelo desenvolvedor durante o *design* do software.

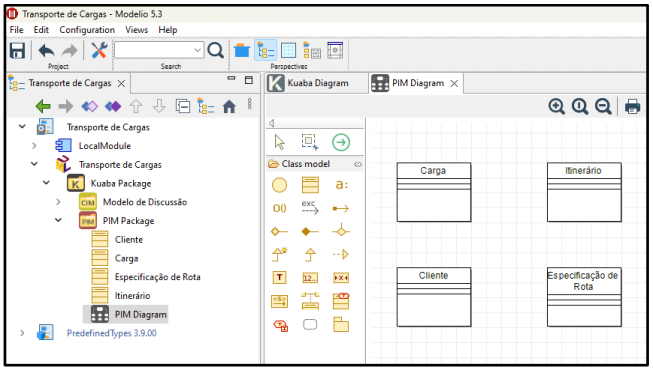

Figura 23. PIM com as classes geradas a partir do CIM

A definição do PIM é feita durante a interação do desenvolvedor com o especialista de domínio, seguindo o princípio da linguagem ubíqua. Durante a definição do modelo, o desenvolvedor utiliza os estereótipos do perfil DDMM para "marcar" os conceitos do domínio de acordo com os padrões táticos do DDD [3]. A Figura 24 mostra o PIM finalizado para o domínio de transporte de cargas com os estereótipos utilizados.

Nesse exemplo, Cliente representa uma pessoa ou empresa e tem uma identidade que é importante para o usuário. Assim, a classe Cliente é marcada com o estereótipo <<*Entity*>> no modelo. Carga também precisa ser distinguível. Na prática, todas as empresas de transporte atribuem ids de rastreamento a cada carga. Assim, a classe Carga também é marcada com o estereótipo <<*Entity*>>.

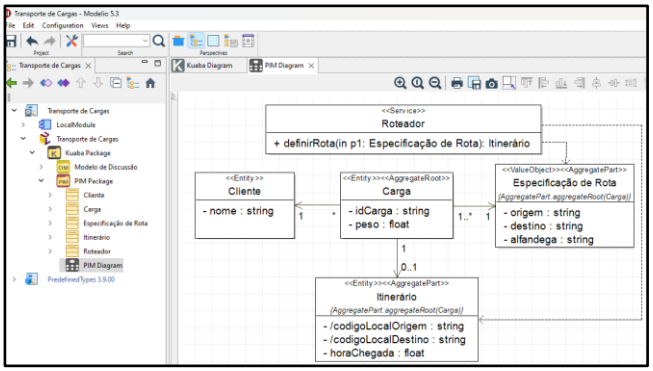

Figura 24. PIM para o domínio de Transporte de Cargas

A Especificação de Rota representa o destino da carga, mas a abstração não depende dela. Ou seja, se há duas ou mais cargas indo para o mesmo lugar, elas podem compartilhar a mesma especificação de rota. Desta forma, essa especificação é um objeto valor e a classe Especificação de Rota é marcada com o estereótipo <<*ValueObject*>>. Uma especificação de rota só precisa ser única no contexto de carga, pois nenhum objeto externo pode vê-la fora do contexto da carga. Assim, Especificação de Rota é uma parte integral de Carga indicando requisitos de entrega e também é marcada com o estereótipo <<*AggregatePart*>>.

Itinerário representa o plano de entrega operacional e também precisa ser único no contexto de carga, sendo a classe

correspondente marcada como <<*AggregatePart*>>. Ambos, Especificação de Rota e Itinerário, fazem parte do agregado Carga, que é a entidade raiz (<<*AggregateRoot*>>), e seus ciclos de vida estão vinculados ao prazo de uma entrega ativa. Roteador é um serviço (<<*Service*>>) que encapsula um mecanismo que encontra um itinerário que satisfaz uma especificação de rota.

5.3 PSM E GERAÇÃO DE CÓDIGO

O modelo específico de plataforma (PSM) é gerado por meio de uma transformação PIM para PSM. Para isto, o desenvolvedor deve clicar com o botão direito do mouse sobre o pacote PIM e selecionar as opções *Kuaba Module/Transform PIM to PSM*. Como resultado, um pacote PSM é criado com os elementos resultantes dos mapeamentos, realizados com base nos estereótipos do DDD presentes nos elementos do PIM, como mostra a Figura 25.

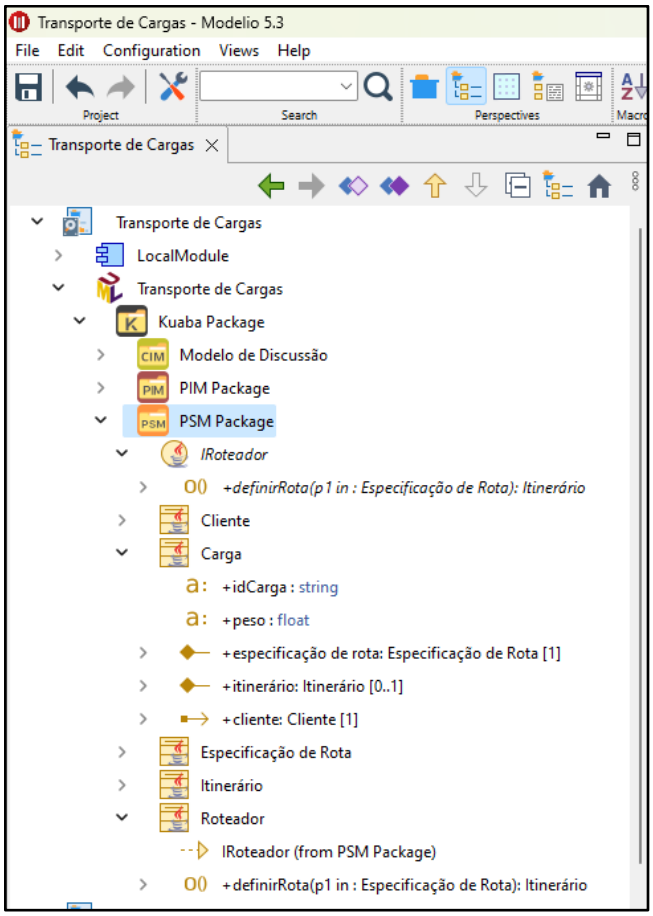

Figura 25. Elementos do PSM gerados automaticamente

Com exceção do elemento Roteador, todos os elementos do PSM são automaticamente marcados com o estereótipo *JavaClass*. Por ser uma interface, Roteador é marcado com o estereótipo *JavaInterface*. Isto permite o módulo *Java Designer* aplicar as transformações nesses elementos para gerar o código fonte na linguagem Java.

Neste exemplo, é possível notar que além da classe Roteador também foi gerada uma Interface Java com a operação abstrata definirRota, uma vez que o conceito Roteador foi marcado como um serviço no PIM. A classe Roteador implementa essa interface e, portanto, possui uma operação concreta definirRota com a mesma assinatura da operação da interface. As relações entre a classe Carga com as classes Especificação de Rota e Itinerário foram transformadas em composições (representadas com losangos na Figura 25), uma vez que o conceito Carga é a raiz de um agregado e, assim, tem a responsabilidade de gerenciar os objetos que compõem esse agregado.

Para gerar o código-fonte em Java o desenvolvedor deve clicar como botão direito do mouse sobre o pacote PSM e selecionar as opções *Java Designer*/*Generate*. As classes serão geradas dentro do arquivo em que está contido o projeto, na pasta "src". O código gerado pode ser visualizado e editado selecionando o elemento desejado no pacote PSM e acessando a aba Java na *view* de propriedades abaixo da área de desenho. Por questões de simplicidade, serão mostrados apenas os códigos gerados para os elementos IRoteador e Roteador. A Figura 26 mostra o código-fonte gerado para a interface IRoteador do pacote PSM. Como a operação pertence a uma interface, ela é abstrata por padrão em Java.

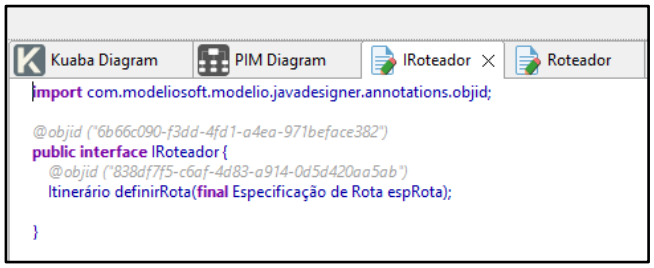

Figura 26. Código-fonte gerado para a interface IRoteador

A Figura 27 mostra o código-fonte gerado para a classe Roteador. Nesse código o módulo *Java Designer* gera um código de retorno padrão que deve ser modificado pelo desenvolvedor.

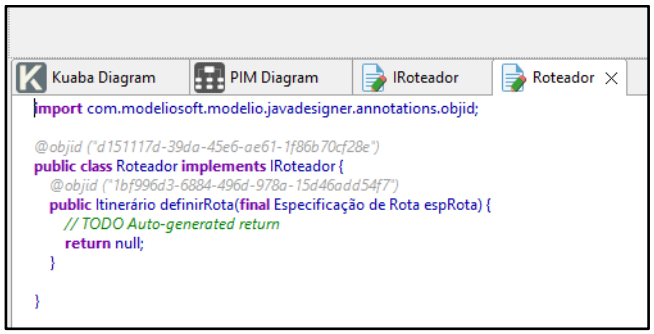

Figura 26. Código-fonte gerado para a classe Roteador.

Os identificadores @objid são inseridos nos códigos pelo módulo *Java Designer* para cada elemento do PSM. Desta forma, os elementos podem ser facilmente identificados, caso seja feita uma engenharia reversa (código para modelo) a partir do código-fonte gerado.

6. TRABALHOS RELACIONADOS

Em [7] é apresentada a abordagem GORE (*Goal-Oriented Requirement Engineering*) para modelar os objetivos organizacionais apoiados por um *Data Warehouse* [5] e relacioná-los a requisitos de informação. De forma semelhante à abordagem proposta neste trabalho, GORE propõe o uso da arquitetura MDA para obter um modelo PIM a partir de um modelo CIM por meio da execução de transformações entre modelos. Porém, a abordagem apresentada é específica para o domínio de *Data*

Warehouse. Os modelos CIM e PIM também são representados com o uso de perfis UML que são utilizados nas regras de mapeamento entre os modelos. No entanto, em GORE, o CIM contém os objetivos e requisitos de informação para o *Data Warehouse* usando uma adaptação (perfis UML) do framework de modelagem i* [22]. A partir de um CIM especificado com o perfil i*, um modelo multidimensional conceitual é derivado em um PIM, por meio de regras de transformação.

Em [20] é apresentada uma ferramenta MDE que também utiliza um perfil DDD, definido pelos autores, para apoiar a geração automática de código. Além do perfil DDD, a ferramenta utiliza um framework *Naked Objects* [16] para a geração automática das camadas de interface do usuário e de persistência. No entanto, a ferramenta não possui suporte a modelos de discussão para obtenção dos conceitos de domínio e nem faz uso de transformações entre modelos para apoiar a geração de código.

A técnica apresentada em [6] foca na transformação de uma especificação de domínio de alto nível, criando artefatos de software seguindo a abordagem DDD, em um modelo de domínio executável. A especificação atual é parcialmente gerada e manualmente escrita em Java. Embora a técnica utilize a abordagem DDD e execute transformações a partir dessa especificação de domínio, como proposto neste trabalho, não inclui suporte a modelos de discussão e não utiliza diretamente os padrões táticos do DDD nas regras de transformação.

7. CONSIDERAÇÕES FINAIS

Este trabalho apresentou a abordagem Kuaba-MDE e o desenvolvimento do módulo Kuaba na ferramenta Modelio para apoiar o uso de modelos de discussão na engenharia orientada por modelos. A abordagem tem como base o DDD e permite aos desenvolvedores elaborar modelos de discussão e conectar esses modelos aos demais artefatos de software por meio de transformações entre os modelos previstos na MDA. Assim, a abordagem é uma contribuição para as pesquisas relacionadas a um dos desafios técnicos apresentados em [2] sobre como usar a MDE para conectar modelos de discussão com artefatos de software.

O módulo Kuaba fornece suporte para criação do modelo de discussão (CIM), a partir do perfil Kuaba, e execução das transformações entre modelos de forma intuitiva. Uma versão inicial do PIM é gerada a partir das decisões registradas no CIM. O módulo facilita o uso do DDD, permitindo que os elementos do PIM possam ser marcados com os estereótipos do perfil DDMM. Esses estereótipos são utilizados nas regras de mapeamento para geração do PSM. Por fim, o uso do módulo *Java Designer* viabiliza a geração automática de código a partir do PSM.

O uso dos *módulos Kuaba e Java Designer* na ferramenta Modelio permite apoiar o desenvolvimento de software desde a modelagem de domínio até a geração de código em uma única ferramenta. Desta forma, este trabalho também contribui com as pesquisas para outro desafio técnico apontado em [2] relacionado à necessidade de alavancar/integrar ferramentas de modelagem flexíveis.

Uma melhoria a ser realizada como trabalho futuro é a implementação de regras de mapeamento específicas para associações bidirecionais. O método atualmente empregado utiliza funções de cópia da API fornecida pela ferramenta Modelio. Isto gera duplicação de associações quando elas são bidirecionais no PIM, ou seja, possuem navegabilidade em ambas as extremidades da associação.

Outro trabalho futuro previsto é o estudo de alternativas para especificar o comportamento interno das operações definidas no PIM de forma que ele possa ser considerado na geração de código. Além disso, está prevista também a realização de avaliações da abordagem em projetos de desenvolvimento de software.

Outra direção promissora para pesquisas futuras é investigar a aplicabilidade da solução desenvolvida para a geração de código em outras linguagens de programação e a integração do módulo Kuaba com o subsistema Kuaba. Essa integração visa permitir a representação de *design rationale* durante a elaboração do PIM em OWL e permitir o processamento computacional desse conhecimento para apoiar consultas sobre o *design*, a realização de inferências e o reúso dos modelos produzidos.

8. REFERÊNCIAS

[1] Bezerra, E. 2015. *Princípios de Análise e Projeto de Sistemas com UML*. Elsevier Editora Ltda.

[2] Bucchiarone, A., Cabot, J., Paige, R. F., and Pierantonio, A. 2020. Grand Challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling 19:5-13.

[3] Cunha, L. E. S. 2023. Transformações de modelos de domínio utilizando a arquitetura MDA para apoiar a geração automática de código. Trabalho de Conclusão de Curso (Graduação), Universidade Federal Fluminense, Instituto de Ciência e Tecnologia, Rio das Ostras.

[4] Evans, E. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[5] Kimball, R., Ross, M. 2002. The Data Warehouse Toolkit. Wiley & Sons, Chichester.

[6] Le, V. V., Be, N. T., & Dang, D. H. 2023. On Automatic Generation of Executable Domain Models for Domain-Driven Design. In 15th International Conference on Knowledge and Systems Engineering (KSE), pp. 1-6. IEEE.

[7] Mazóm, J-N., Pardillo, J., Trujillo J. 2007. A Model-Driven Goal-Oriented Requirement Engineering Approach for Data Warehouses. ER Workshops, LNCS 4802, pp. 255-264. Springer-Verlag Berlin Heidelberg.

[8] Medeiros, A. P. 2020. Pesquisando Soluções para Desafios em MDE com Kuaba. In Anais do II Workshop em Modelagem e Simulação de Sistemas Intensivos em Software pp. 11-15. SBC.

[9] Medeiros, A. P. and Schwabe, D. 2008. Kuaba approach: Integrating formal semantics and design rationale representation to support design reuse. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 22:399-419.

[10] Medeiros, A. P., Schwabe, D., & Feijó, B. (2005). Kuaba ontology: design rationale representation and reuse in model-based designs . In Conceptual Modeling–ER 2005: 24th International Conference on Conceptual Modeling, Klagenfurt, Austria, October 24-28, 2005. Proceedings 24 pp. 241-255. Springer Berlin Heidelberg.

[11] Nunes, T. R.; Medeiros, A. P. 2009. KSE – ferramenta de apoio à captura e representação semiautomática de design rationale em Engenharia de Software. In: Anais do XXIII Simpósio Brasileiro de Engenharia de Software (SBES) - XVI Sessão de Ferramentas, Fortaleza.

[12] OMG. 2010. Business Process Model and Notation. Disponível em: https://www.omg.org/bpmn/ Acesso: 08/04/2024.

[13] OMG. 2014. OMG MDA Guide rev. 2.0. Disponível em: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf Acesso: 07/06/2023.

[14] OMG. 2016. Meta Object Facility (MOF). Disponível em: https://www.omg.org/spec/MOF. Acesso: 26/12/2023.

[15] OMG. 2017. Unified Modeling Language. Disponível em: https://www.omg.org/spec/UML. Acesso: 08/04/2024.

[16] Pawson, R. 2004. Naked Objects, Phd thesis, Dublin: Trinity College.

[17] Rademacher, F. Sachweh, S. and Zündorf, 2018. Towards a UML profile for domain-driven design of microservice architectures. In Software Engineering and Formal Methods: SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers 15. 230-245. Springer International Publishing.

[18] Schmidt, D. C. 2006. Model-driven engineering. IEEE Computer, 39 (2), 25-31.

[19] Selic, B. 2003. The pragmatics of model-driven development. IEEE Software, v. 20, n. 5, p. 19–25.

[20] Soares, S. A., Brandão, M., Cortés, M.I., Freire E.S.S. 2015. Dribbling complexity in model driven development using Naked Objects, domain driven design, and software design patterns. Latin American Computing Conference (CLEI), Arequipa, Peru, pp. 1-11, doi: 10.1109/CLEI.2015.7360022.

[21] W3C. 2012. Web Ontology Language (OWL). Disponível em: https://www.w3.org/OWL/. Acesso: 26/12/2023.

[22] Yu, E. 1997. Towards modeling and reasoning support for early-phase requirements engineering. In: RE 1997, pp. 226–235.